Reimplementing a Ruby on Rails Site in Yesod

CS 557: Functional Languages @ Portland State University
Dylan Laufenberg, winter 2019

1 Introduction

For the last seven years or so, I've been developing and expanding a Ruby on Rails-powered website for a Sacramento, CA soup kitchen named Sharing God's Bounty. I've had mixed feelings about my choice of Ruby on Rails over the last half-dozen years, so this final project seemed to be an excellent opportunity to experiment with a significantly different Web technology stack than any other that I've ever used. I've implemented a narrow, vertical slice of functionality to experience and experiment with Haskell across many aspects of server-side code for the Web. I use the Yesod web framework for my project, and I provide comparisons to my Ruby on Rails experience as appropriate through this report. I followed the quick start instructions here, which provided me with Yesod 1.6.0.3.

Yesod's documentation comes primarily from the Yesod book, so a large portion of my work for this project has simply been reading through the book to understand how to utilize this highly sophisticated, mature framework. I don't attempt to introduce or define every concept in Yesod here. Instead, I highlight particularly noteworthy applications of Haskell to the domain of Web application development and discuss how I use Yesod to build out the features I develop for this final project. In particular, I assume the reader knows about Haskell features such as Template Haskell, QuasiQuotes, and language pragmas either through prior knowledge or by reading the Haskell section of the Yesod book.

2 Resources and type-safe URIs

To represent **routes**, which are mappings from URIs to server-side code, Yesod uses what it calls resources. A **resource** is a `data` constructor that acts as the canonical name of the route in views. For instance, the default site scaffold includes a static route of the form:

```
/static StaticR Static appStatic
```

This line specifies a `/static` route, a resource constructor named `StaticR` as the name of the route, and a subsite named `Static` whose function `appStatic` will handle requests on this route. In order to create a link to a static file, e.g. an image served as part of a template, we invoke the `StaticR` constructor with an appropriate argument. The `Static` subsite generates "static file identifiers" at compile time [1]. When we insert a link using a resource constructor:

```
<img ... src="@{StaticR img_logo_png}" ... />
```

Yesod generates a correct link to the static resource at compile-time. In this case, the user's browser receives:

```
<img ... src="http://localhost:3000/static/img/logo.png?etag=GVVUzjtL" ... />
```

There are two primary benefits to this approach, both of which leverage Haskell features from class (albeit at a very advanced level).

Check links at compile time Yesod uses resources to generate correct links at compile-time, which allows it to check whether said links are valid and raise compile-time errors if any problems are detected. In this case, the Static subisite checks the identifier `img_logo_png` against all generated static file identifiers. If it finds a match, it inserts the correct link. If not, then GHC raises an error. For instance, if we mistakenly specify a JPG logo:

```
<img ... src="@{StaticR img_logo_jpg}" ... />
```

GHC will detect the issue for us and (rather obtusely) notify us of the issue with a compile-time error:

```
• Variable not in scope: img_logo_jpg :: Route Static
• Perhaps you meant ‘img_logo_png’ (imported from Import.NoFoundation)
|
```

```
163 |             $(widgetFile "default-layout")
    |               ^^^^^^^^^^^^^^^^^^^^^^^^^^
```

In effect, Yesod automatically tests every internal link at compile time, without the need to manually write tests against the generated HTML. (Having written Bounty's Rails front-end largely via test-driven development, I can attest that Yesod saves me an inordinate amount of time!)

Refactor routes with confidence Generating links at compile time also eliminates an entire class of common and nefarious errors: broken links. Updating or moving routes is a common occurrence in Web application development, even with careful planning. Yesod makes this safe and automatic by generating all such links from resource constructors at compile time. to change the route across the entire application, all we need to do is change the route definition:

```
/dynamic StaticR Static appStatic
```

And all of our generated links to that route will automatically update when we recompile:

```
<img ... src="http://localhost:3000/dynamic/img/logo.png?etag=GVVUzjtL" ... />
```

3 The basics

I'm not sure what should go here, exactly, but the chapter on the basics definitely deserves some discussion AS IT PERTAINS TO MY CODE. I don't think there's much value in covering things I didn't write.

4 Shakespearean Templates

Topics:

- Hamlet: went pretty smoothly, Initially had to cut out lots of features as NYI.
- Lucius/Cassius: oh, man. Whole subsection(s), whole big thing.

5 Reflections after the basics

Reflect on Yesod from the perspective of someone who's just made it through the basics. In particular, TinyOS's observation about being hard for beginners. I feel like I have to be an expert to do almost anything. Every piece of code feels like it's written for experts. And the lack of an accessible API reference (with good documentation) so far seems to be frustrating me.

References

[1] Michael Snoyman. *Developing Web Applications with Haskell and Yesod*. 1st. O'Reilly, 2012.